dccautomation Documentation

Release 0.1.0

Rob Galanakis

July 10, 2014

Contents

The dccautomation library facilitates the external control or automation of any application that hosts a Python interpreter. This is very useful when writing automated tests or systems against a program, such as Autodesk Maya, that does not behave like a normal Python interpreter. A client process (usually a normal Python interpreter) can connect to a server process (the application), and have the Python interpreter execute and evaluate Python commands.

This is conceptually a simple and already served RPC mechanism, but the complications of various DCC packages makes a package like `dccautomation` hugely beneficial, since it is both extensively unit and production tested.

- **dccautomation** on GitHub: https://github.com/rgalanakis/dccautomation
- **dccautomation** on Read the Docs: http://dccautomation.readthedocs.org/
- **dccautomation** on Travis-CI: https://travis-ci.org/rgalanakis/dccautomation
- **dccautomation** on Coveralls: https://coveralls.io/r/rgalanakis/dccautomation

CHAPTER 1

Usage

Using dccautomation is very simple. First, you must create or find a config for your application. Then, you can run the dccautomation client and server in one of two ways: with a paired client and server, or with a server running on a well-known port.

1.1 Using a Config

Using a config (instance of the dccautomation.configs.Config class) is very simple. First, look in the dccautomation.configs module for a configuration for your application. New configurations are added regularly.

If you cannot find one there, you can subclass dccautomation.configs.Config and override the necessary methods.

In either case, this is all the code you have to write to get automation working. You just pass this configuration around and let dccautomation do all the work.

1.2 Paired client and server

The most common and useful way to use this library is to bootstrap a server process and pair it with its own client. This ensures nothing else can interfere with the server (remember, the server is usually a very stateful application, you wouldn't want state magically changing between client calls!). This bootstrapping also allows you to have many server instances running, because they will each be communicating over a unique port. You can do this by running in your client process (normally a standard Python interpreter):

```
import dccautomation
cfg = MyAppConfig()
svrproc = dccautomation.start_server_process(cfg)
client = dccautomation.Client(svrproc)
client.exec_('import mycode; mycode.do_stuff()')
```

Using a paired client and server is most useful when doing automated testing, allowing you to run code that doesn't work in a normal Python interpreter. It can also be useful when doing batch processing, by starting up several instances of the application into a worker pool.

1.3 Well-known server and many clients

Another useful scenario is to start a server on a background thread inside of an already running application, using a well-known port. Any number of clients can then connect to it. You can do this by running the following in your application/server process:

```
import dccautomation
dccautomation.start_inproc_server(MyAppConfig(), 9023)
```

Then in your client process, you can run the following:

```
import dccautomation
client = inproc.start_inproc_client(MyAppConfig(), 9023)
client.exec('import mycode; mycode.do_stuff()')
```

Note that your client and server need to know what port to communicate on beforehand (we use `9023` in the previous examples). This also means we can only have one server/application process running.

You can also set the `DCCAUTO_INPROC_PORT` environment variable to set the port (usually before launching the process), instead of hard-coding it to a number. Do not pass in a port number, and the environment variable will be read.

Using a well-known server and one or more clients can be useful when exploring how code works. You can open up a GUI session of your application, start a server, run some code from the client process that manipulates the application, and visually inspect the result afterwards.

It can also be useful when communicating between applications. You can have Maya tell your game engine to update based on some change in Maya, and you can have Maya update based on some change in the game engine. To achieve this, you could do something like this:

```
# In your game engine
import dccautomation
dccautomation.start_inproc_server(GameEngineConfig(), 9024)
maya = dccautomation.start_inproc_client(MayaConfig(), 9025)
maya.exec_('import pymel.core as pmc')

# In Maya
import dccautomation
dccautomation.start_inproc_server(MayaConfig(), 9025)
game = dccautomation.start_inproc_client(GameEngineConfig(), 9024)
game.exec_('import levelleditor')
```

CHAPTER 2

Writing tests that run in your application

One of the major benefits of dccautomation is the dccautomation.RemoteTestCase class. You can subclass this class, and your test methods will run in your application. This allows you, for example, to write normal-looking test code, and then use standard Python tools (like nosetests) to run your code. For example, you could have the following test code:

```
import dccautomation, my_configs
try:
    import pymel.core as pmc
except ImportError:
    pmc = None

class SillyPymelTests(dccautomation.RemoteTestCase):
    config = my_configs.MayaConfig

    def testFindsActive(self):
        jnt = pmc.joint()
        self.assertEqual(jnt.type(), 'joint')
```

Then, you can run the tests in whatever fashion: from your IDE, through nose or any test runner, whatever. Under the hood, RemoteTestCase works some magic and your code is executed inside your application.

CHAPTER 3

Design

As stated previously, conceptually `dccautomation` is a simple RPC system. In practice, setting up an RPC system using applications that host Python is not trivial. They have particular startup mechanics, are slow to start up, have special environment setups and libraries, and other considerations. Many people need to write code in these environments, but lose the benfit of modern tools or practices. If you've ever tried to do Test Driven Development in Maya, you have run into these issues!

So we created `dccautomation` to solve the needs of:

- Write automated tests that transparently run in custom applications.
- Have a way for a pure-Python application to use a custom application for special data processing (think something like an exporter that runs in a standard Python interpreter, that when you export will open up Maya behind the scenes to export the model).
- Parallel batch processing.

Internally, `dccautomation` uses PyZMQ. In the future, the protocol mechanism may be configurable, or changed to a pure-Python mechanism, to eliminate comnpatibility issues.

CHAPTER 4

Authors

The primary author is Rob Galanakis, rob.galanakis@gmail.com. The initial concepts of `dccautomation` were developed during my time at CCP Games. I give special thanks to my former colleagues there for proving that given the right opportunity and tools, people can improve and excel.

CHAPTER 5

Useful API Members

The dccautomation.configs module is the primary module clients will need to use. It contains the Config base class, and a number of pre-defined subclasses for popular DCC applications and platforms. You can get a config instance by its class name using config_by_name().

class dccautomation.configs.**Config**
Configuration for a controllable process.

cfgname()
Return the configuration name. Used when many configs should share the same name (such as various OS flavors of a DCC app).

dumps(*data*)
Dump the data into a string and return the string (bytes). Defaults to json.dumps(data).encode('utf-8').

exec_context()
Return a callable will run exec and eval. Useful if the exec and eval must occur on a certain thread.

loads(*s*)
Load data from a string (bytes). Defaults to json.loads(s.decode('utf-8')).

popen_args()
Return a list of command line arguments used to start a process and have it run an automation server.

dccautomation.configs.**config_by_name**(*name*)
Return the first config type that has a member/type name matching name.

The dccautomation.configs module also contains a number of pre-defined configs clients can use.

class dccautomation.configs.**CurrentPython**
The current executable. Assumed to be a valid Python interpreter.

class dccautomation.configs.**SystemPython**
The system python interpreter (what you'd get typing "python" from the command line).

class dccautomation.configs.**Maya2015OSX**

dccautomation.configs.**Maya**

Finally, the dccautomation.RemoteTestCase class is very useful for clients who wish to use dccautomation to run automated tests.

class dccautomation.**RemoteTestCase**(*methodName='runTest'*)
Subclass for test classes which should execute in the DCC process. This is sort of magical, and you can generally stay out of it by overriding this class and some of its attributes:

•config: A subclass of dccautomation.configs.Config. Do *not* use an instance of it.

- reload_test: If True, reload the test file before running each test.
- **cache_client: If True, use one client for all test methods.** If False, use a new client for each test. Clients are created through the `create_client()` method.
- start_proc: If True, start a server process before creating the client.
- reload_modules: If supplied, reload these modules before running a test. It must be a module instance.

Most of this behavior is used in the `create_client()` method. Override this method for advanced usage.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

d